BASIC COMPILER

FROM ADVANCED SOFTWARE

FOR THE TI 99/4A COMPUTER

REQUIRES EXTENDED BASIC
AND MEMORY EXPANSION

CONTENTS

INTRODUCTION

Most TI 99/4A users are confronted with two alternatives when writing programs. On one hand, programming in BASIC is relatively simple and graphics and sound are easily accessible. As is well known, however, the resulting graphics are much slower than arcade type action. On the other hand, programming in ASSEMBLY solves the problem of speed, but learning the language itself is difficult and programming in it can be frustrating and time consuming.

The graphics of BASIC are slow due mainly to the fact that when the program is running, the computer must read each statement, interpret it, check it for errors, and then, finally, call the neccesary built-in subroutines to execute it. The built-in program that does all of this is called a BASIC INTERPRETER.

Writing a program in ASSEMBLY language is somewhat equivalent to specifying a list of subroutine calls. When the computer executes the program, it will call those subroutines as instructed, not expending time in interpreting or checking for errors. The progammer has to understand many details about the internal functions of the computer and is fully responsible for what happens. There is no easy way to find the errors and correct them.

A more reasonable approach to programming is using a compiling system, that is, a system to translate the BASIC statements into subroutine calls and save them. When the compiled program - i.e., a list of subroutine calls translated from BASIC statements - is called, the subroutines are executed as in an assembly program. There are therefore two steps involved in the process: compiling and running.

The ADVANCED SOFTWARE BASIC COMPILING SYSTEM (ASBCS) is a set of two programs: an EXTENDED

BASIC program, the COMPILER, which transforms BASIC statements into number codes, and an ASSEMBLY program, the RUNNER, which executes operations by calling a specific subroutine for each of those number codes. The RUNNER can be called by any regular EXTENDED BASIC program. The statements to be compiled are inserted as BASIC lines after the last lines of the COMPILER itself. The subroutines provided allow you to execute the most important operations supported by EXTENDED BASIC and, in addition, other operations only available with ASSEMBLY. Graphics are controlled by assigning values to some specific variables. This results in the big advantage of a compiled program over a regular EXTENDED BASIC program: hundreds of times faster graphics operations. One disadvantage is that you have to learn how these variables control the graphics and keep track of them when programming. This is also true for the rest of the computer operations. The programs produced by the COMPILER are much faster but less easy to write than regular EXTENDED BASIC programs.

Programming with the ASBCS provides you with very efficient ways to use the computer's memory. This is due to the fact that you work more directly with the memory and that the number of reference tables used is reduced. Also, much memory space is saved by working with some of the variables as one- and two-byte integer numbers as compared with the rigid eight-byte real number format used in EXTENDED BASIC.

With a compiled program you can perform the most important operations available with the TI 99/4A, such as arithmetical operations, graphics, including sprites, and logical operations. The display is originally set to the GRAPHICS mode, as in EXTENDED BASIC. However, with a single instruction in the compiled program, it can be changed to TEXT, MULTICOLOR or BIT-MAP mode.

Editing is done as with a regular EXTENDED

BASIC program. Once your program is written, the COMPILER compiles it, that is, assigns number codes to each statement and stores these codes in memory. During this compiling process, the program will check the statements and, if it finds an error, will issue an error messsage and stop running. The error message will also indicate the number of the line where the error occurred.

After your program is compiled free of error, you can then run it by calling the RUNNER with an EXTENDED BASIC program or command. If an error occurs at this stage, the program will once again issue an error message indicating the line in the compiled program where the error occurred. This means that the capability to diagnose errors available in EXTENDED BASIC is maintained while running the compiled program.

As you can see, the ASBCS is a compromise between the speed and power of the ASSEMBLY and the simplicity of BASIC. Of course, you can expend a considerable amount of money in hardware and software, and/or the time to learn a new programming language of a conventional and more powerful system such as PASCAL or FORTH. But, if you want to take advantage of the equipment and the knowledge you have accumulated so far, the ADVANCED SOFTWARE BASIC COMPILING SYSTEM (ASBCS) gives you a chance to do so. To make the most advantage of your computer, you should write a part of your program in EXTENDED BASIC and write compiled subroutines using the ASBCS for the operations where EXTENDED BASIC is not fast enough for you.

This manual is divided into six chapters. Chapter one describes the characteristics of the variables that can be used. In chapter two, the statements' syntax is presented, including the types of errors that can result when executing them. Chapter three deals with compiling. The instructions to link to a compiled program are

given in chapter four. In chapter five, character display and sprites are covered. Two examples are discussed in chapter six. Instructions for handling graphics in the BIT-MAP, MULTI-COLOR and TEXT modes are given in the Appendix. In addition, the Appendix includes some fundamental concepts and data about the computer memory use and operations.

1. VARIABLES

The variable names in the program to be compiled are restricted to the letters from C through V for single numeric variables. Names for one-dimensional array numeric variables are restricted to W(),X() and Y(). The only name permitted for a one-dimensional array string variable is Z$(). The subscript of Z$() and Y() can not be zero.

Multidimensional array variables and single string variables are not supported.

The subscript of W() must have an integer value from zero through 16383 and the values of its elements can be any integer from 0 through 255.

The subscript of X() must be an integer value from zero up to 12284 ,depending on the size of the EXTENDED BASIC program that is calling the compiled program, and the values of its elements can be any integer from -32768 through 32767.

The values of the single variables C through V can also be any integer from -32768 through 32767.

The arrays Y() and Z$() are regular BASIC variables and have to be dimensioned in the calling program. These are also the only two of all the above mentioned variables that can be modified by the EXTENDED BASIC calling program.

2. SYNTAX

With the ASBCS you can compile the most important and useful EXTENDED BASIC statements. Most of the statements generate the same or similar operations when executed in EXTENDED BASIC or when executed by the RUNNER as compiled statements, so you can check the logic of your program before compiling it. However, a few statements, after being compiled, will generate operations not supported by EXTENDED BASIC when executed by the RUNNER, and in those cases the action generated is meaningless when executed in EXTENDED BASIC. For example, the statement LET B=BITMM, when executed by the RUNNER, sets the display to BIT-MAP mode. However, if run in EXTENDED BASIC, B and BITMM are two dummy variables.

The subcripts of the arrays have to be integer single variables. For examle, Y(N) and Z$(Q) can appear in a statement, but W(1), Z$(Y(C)) and X(100) will generate a compiling error. In addition, integer numbers from 0 through 31 can appear in any statement.

The valid statements are listed below. An explanation of the statement is given only when the operation generated is different from EXTENDED BASIC.

ARITHMETIC OPERATIONS

numeric variable=item-arithmetic operator-item

where item is a numeric variable or an integer number from 0 through 31 and arithmetic operator is either +,-,*,/ or ^. Examples of acceptable statements of this type are: D=Y(B)+J and X(C)=F*20

first numeric variable=second numeric variable

Examples: W(F)=R N=C

numeric variable=number

where number is an integer from -32768 hrough 32767. `Examples: K=10000 Y(J)=-301

numeric variable=RND

RND is a random generated number from -32768 through 32767.

first numeric variable=ABS(second numeric variable)

Non-integer values are rounded out before being assigned to an integer variable.
For Example W(R)=31/4 makes W(R) equal to 8. The numbers can be raised only to a positive integer or zero power.

ACCEPT AT(row,column):numeric variable

where row is a numeric variable or an integer from 1 through 24 and column is a numeric variable or an integer from 1 through 24 in GRAPHICS mode, or through 40 in TEXT mode. Assigns the ASCII code of the key pressed to the variable and displays the character.

BREAK

Causes program to halt when encountered. Execution continues after any key is pressed.

CALL CHAR(character code,Z$())

CALL CLEAR

CALL COLOR(row,column,color code)

assigns color to box in MULTICOLOR mode. See Appendix for insructions.

CALL COLOR(character code,Z$())

assigns sixteen colors to character in BIT-MAP mode. See Appendix for instructions.

CALL JOYST(key unit, x return, y return)

Same as in EXTENDED BASIC but additionally stores the status in the variable S. The status information is used to check if the fire button was pressed.

CALL KEY(key unit,return variable,status variable)

CALL MAGNIFY (magnification factor)

CALL SCREEN (color code)

changes the screen color in Graphics mode and the foreground and background colors in TEXT mode. See Appendix for TEXT mode handling.

DISPLAY AT(row,column):Z$()

row and column as in ACCEPT.

END

returns control to EXTENDED BASIC. And also returns the line number of the END statement plus 10 in the EXTENDED BASIC variable LINE.

IF relational expresion THEN line number

relational expression compares two variables or a variable and an integer number from 0 through 31.

FOR control variable=initial value TO limit STEP increment

same as in EXTENDED BASIC but the control variable must be an single variable (C through V). The initial value, the limit and the increment can be numeric variables or integers from 0 through 31.

GOSUB line number

GOTO line number

is the fastest operation, so go to ....

LET B=CPTCP*first address*second address*length

reads a memory segment starting at CPU RAM first address and writes the segment starting at CPU second address; length is the number of bytes in the segment. For example, if the current values of C,D and E are 0, 10000 and 2000, the statement

LET B=CPTCP*C*D*E

reads the CPU RAM segment from address 0 through 1999 and writes it on the CPU RAM segment from address 10000 through 11999. Caution: improper use of this statement may cause the computer to "lock up". See the MEMORY USE section of the Appendix.

LET B=CPTVD*VDP address*CPU address*length

writes a memory segment from CPU RAM to VDP RAM.

LET B=VDTCP*VDP address*CPU address*length

writes a memory segment from VDP RAM to CPU RAM.

LET B=STTCP*CPU address*ZZ(single variable)

writes the string Z$(single variable) on CPU RAM. For example, if the values of X(N) and Z$(P) are 10000 and "ABC" respectively, the statement

```
LET B=STTCP*X(N)*ZZ(P)
```

will make the bytes from 10000 through 10003 equal to 3,65,66 and 67.

LET B=CPTST*CPU address*ZZ(single variable)

makes Z$(single variable) equal to a segment of CPU RAM. For example, if the bytes from CPU address 14000 through 14004 are 4,65,65,65 and 65, and the value of Y(D) is 14000, the statement

```
LET B=CPTST*Y(D)*ZZ(K)
```

will make Z$(K) equal to "AAAA".

LET B=GRAPM

sets the display to GRAPHICS mode.

LET B=BITMM

sets the display to BIT-MAP mode.

LET B=TEXTM

sets the display to TEXT mode.

LET B=MULTM

sets the display to MULTICOLOR mode.

LET B=RESET

clears the subroutine call and FOR-NEXT stacks. The first time you link to a compiled program this must be the first statement executed. You can leave the compiled program and return to it at any line. If this statement is not executed when returning from EXTENDED BASIC, the sequence of GOSUB-RETURNs and FOR-NEXTs is mantained. This means that you can treat your EXTENDED BASIC program as a subrotine of your compiled program.

NEXT control variable

PRINT ,,,,

scrolls the screen up a number of lines equal to the number of commas plus 1.

RANDOMIZE /seed/

the optional seed is an integer from -32768 through 32767.

RETURN

3. COMPILING

Load the COMPILER from cassette or disk. Write the program you want to compile after the last line of the COMPILER. The program to be compiled can have up to 680 lines. Next, input the command

>RES initial line

If you want the first line of your program to be 2400, "initial line" must be 100. Otherwise, subtract 2400 from the number you want for the first line and add the difference to 100 to calculate "initial line". For example, if you want the first line to be 3000, "initial line" should be 700. The highest and lowest values allowed for the line numbers are 2400 and 9190.

You can check the syntax and logic of your program by running it, if the subscripts of the arrays W() and X() aren' too high . To run, input the command

>RUN line number

where "line number" is the first line of your program. Even if you can not take or choose to skip this step, your program is now ready to be compiled.

Input the RUN command. When the input prompt (?) appears type COMPILE (or just CO) and press <ENTER>. Next input the numbers of the first and last lines you want to have compiled. The COMPILER will then start working and will display the number of the line being compiled.

If the COMPILER finds an error, it will issue an error message indicating the type of error and the line number where it occurred, and will stop running. The error messages are the following:

BAD LINE NUMBER
BAD STATEMENT
MEMORY FULL
BAD VALUE

See the Appendix for descriptions of these messages. Correct the error and begin compiling again, starting with the line just corrected. Repeat the process until the program is free of error.

When the compiling is done, the input prompt will appear again. If you want to save the compiled code type SAVE (or just SA). You will then be asked for the first and last line numbers of the program and the title you wish to assign to it. The compiled code is saved as sequential and internal records which are 192 in length. Each record contains one string 191 long.

After the code is saved, the prompt will appear. If you want to load the RUNNER, or any any compiled program previously saved by the COMPILER, input LOAD (or just LO). When loading the RUNNER, the first and last addresses of the memory segment where it is stored and the title RUNNER will be displayed. Type Y if you wish to continue loading, or N otherwise. When loading a program, its first and last line numbers are displayed instead of addresses but otherwise the procedure is the same as when loading the RUNNER.

At this point it is advisable to list your program on paper if you have a printer and then save it on diskette or cassette tape. Of course, the COMPILER and your program are now saved as just one program. To stop the COMPILER just type STOP (or just ST).

4. RUNNING

Now you should see what your compiled program does by linking to the RUNNER while the COMPILER is running, assuming that you have previously loaded the RUNNER. Input LINK (or just LI) and you will be asked for the line number of the first statement to be executed. You can specify any compiled line number. You must be careful not to specify a line number not compiled because the results will be unpredictable. After this, the RUNNER takes over and executes your compiled code.

If you press <BACK> (FCTN 9), control is returned to EXTENDED BASIC and will generate RUNNER error number 12. Pressing <QUIT> will do the same that in EXTENDED BASIC but your compiled code will remain in memory.

If an error occurs, control is passed back to EXTENDED BASIC and an error message will be displayed including the line number at which it occurred (See the Appendix for an explanation of error messages). If your compiled program hasn't written on top of the COMPILER, the input prompt will appear again and you can proceed to correct the error. If the COMPILER code has been altered, it is best to reload it from cassette or diskette. If your compiled program runs free of error, and a compiled END statement is found, control is also returned to EXTENDED BASIC and the prompt appears again.

In order to link your EXTENDED BASIC program to your compiled program, you have to include the LINK subroutine in it. The subroutines LINK, SAVE and LOAD are supplied together as one program saved following the COMPILER. With LOAD and SAVE you can load and save compiled programs or data, while your EXTENDED BASIC program is running. Instructions on how to use them are given in the program inserted as REMARKS.

5. GRAPHICS

There are two major differences between a regular EXTENDED BASIC and a compiled program with respect to the screen display. First of all, a compiled program does graphics operations mainly by assigning values to specific elements of the array W(), instead of calling subroutines. Secondly, the compiled program can access TEXT, MULTICOLOR and BIT-MAP display modes, while GRAPHICS is the only mode available in EXTENDED BASIC. In this chapter, graphics operations in GRAPHICS mode are discussed. See thee Appendix for the information about the other display modes.

5.1 Character Definition

In a compiled program, the characters are defined by the statement

CALL CHAR(character code,Z$())

The length of the string Z$() can be from 2 through 240 and defines up to 15 consecutive characters. If the length is not an even number, the last character of the string is ignored. If the string is less than 16 characters, the remaining portion of the pattern is not changed. For example, if character 128 was previously defined as "FFFFFFFFFFFFFFFF", C is 128 and Z$(D)="00000", execution of the statement CALL CHAR(C,Z$(D)) will make the pattern of character 128 "0000FFFFFFFFFFFF".

5.2 Character Color

The colors for the characters are specified by W(2064) through W(2077). Thus, W(2064) specifies

the colors of the characters 32 through 39, W(2065) specifies colors for characters 40 through 47 an so on. W(2077) specifies colors for characters 136 through 143. To assign a color to a set of characters, make the corresponding W() equal to the foreground color code times 16 plus the background color code minus 17, or 16*FCC*16+BCC-17. For example, if C is 2064, the statement W(C)=23 makes the colors of the first character set black on cyan, since 2 (black) times 16 plus 8 (cyan) minus 17 is 23.

5.3 Character Display

W(0) through W(767) specify the characters displayed on the 768 positions of the screen. W(0) through W(31) make up the first row, W(32) through W(63) the second row and so on. W(736) through W(767) make up row 24. To display a character, make the W() corresponding to the desired position equal to the character code plus 96. For example, if you make W(34) equal to 161 (65+96), character 65 (normally the letter A) will appear on the third column of the second row.

5.4 Sprite Position, Pattern and Character

W(768) through W(879) specify the vertical and horizontal positions, character value and color (these four are called the sprite attributes) of the 28 sprites. W(768) through W(771) are the dot-row, dot-column, character code plus 96 and color code minus one of sprite #1. For example, making these four values 40,50,161 and 15, will set sprite #1 to dot-row 40, dot-column 50, character value 65 and white color. W(772) through W(775) are the attributes of sprite #2, and so on up to W(876) through W(879).

As sprites move, the the elements of W() that specify positions, change automatically. In order to move sprites with the compiled program, they have to be initially defined in EXTENDED BASIC with a velocity (even if the velocities are zero).

5.5 Sprite Velocity

The velocities of the 28 sprites correspond to the elements W(1920) through W(2029). W(1920) and W(1921) are the row- and column-velocity of sprite #1. A positive velocity is a value from zero through 127. A negative velocity is W() minus 256. For example, if W(1920) is 3 and W(1921) is 240, the row- and column-velocity of sprite #1 are 3 and -16 respectively. W(1924) and W(1925) are the velocities of #2, W(1926) and W(1927) the velocities of #3 and so on up to W(2208) and W(2209) for #28.

6. EXAMPLES

The following example demonstrates the speed with which a compiled program can modify the screen display. The screen content is saved on memory and you can subsequently clear or restore the screen content by pressing 3 or 5. It will return to EXTENDED BASIC by pressing any other key.

```
2400 LET B=RESET
2410 C=-24576
2420 D=768
2430 LET B=VDTCP*0*C*D
2440 CALL KEY(1,I,S)
2450 IF S=0 THEN 2440
2460 IF I=8 THEN 2510
2470 IF I=10 THEN 2490
2480 END
2490 LET B=CPTVD*0*C*D
2500 GOTO 2440
2510 CALL CLEAR
2520 GOTO 2440
```

Type the statements after line 2390 of the COMPILER, compile from 2400 to 2520 and link to line 2400. Then see how fast you can clear and restore the screen. Indeed, this is how a program can change frames so quickly that it creates the impression of movement.

Line 2400 clears both the subroutine call and FOR-NEXT stacks. In this case it is not really necessary because the program doesn't use the statement GOSUB or FOR-TO, but it should always be included in order to get into the habit of doing so.

Lines 2410 through 2430 save the screen content (VDP RAM from zero through 767) on CPU RAM starting at address -24576. Lines 2440 and 2450 halt the program until any key is pressed.

Lines 2460 and 2470 select the operation. Line 2480 returns control to EXTENDED BASIC. Line 2490 restores the screen content. Line 2510 clears the screen. After returning to EXTENDED BASIC, the value of the variable LINE should be 2490. Stop the COMPILER and input the command PRINT LINE to check.

This next example demonstrates the use of sprites in a compiled program. One sprite moves horizontally at the top of the screen with a randomly selected speed. A second sprite moves from the bottom to the top of the screen after any key is pressed. If a coincidence occurs, both sprites will stop until you press a key again. Pressing <BACK> (FCTN 9), will return control to EXTENDED BASIC.

```
2400 LET B=RESET
2410 C=768
2420 D=769
2430 E=772
2440 F=773
2450 H=1921
2460 I=1924
2470 K=129
2480 CALL CLEAR
2490 L=RND
2500 L=ABS(L)
2510 W(H)=L/K
2520 W(E)=180
2530 W(I)=0
2540 CALL KEY(5,R,S)
2550 IF S=0 THN 2540
2560 W(I)=156
2570 IF W(E)>16 THEN 2650
2580 M=W(D)-W(F)
2590 M=ABS(M)
2600 IF M>8 THEN 2650
2610 W(I)=0
2620 W(H)=0
2630 BREAK
2640 GOTO 2480
2650 IF W(E)<6 THEN 2520
2660 GOTO 2570
```

Type the program after line 3990. Modify line 1210 of the COMPILER to

```
1210 CALL SPRITE(#1,65,2,10,10,0,0,#2,66,2,182,
128,0,0)::CALL LINK("RUNNER",Y(),Z$(),LINE,ERR,
VAR())
```

Compile and link to 2400. Now try to intercept A with B by pressing any key. Press <BACK> to

return to the COMPILER. The message

```
# BACKPOINT IN line number
```

will be displayed.

Lines 2490 through 2510 assign a random column velocity to sprite #1. Line 2530 makes the row velocity of sprite #2 zero. Lines 2540 and 2550 wait until any key is pressed. Line 2560 makes the row velocity of sprite #2 equal to -100. Lines 2570 through 2600 check for coincidence between the sprites. Lines 2610 and 2620 stop the sprites. Line 2630 halts the program until any key is pressed. Line 2650 checks if sprite #2 is close to the top of the screen.

APPENDIX

APPENDIX A

VALID STATEMENTS

"var" stands for numeric variable or 0 through 31.

array subscripts must be single integer variables.

```
  ARITHMETIC OPERATIONS
    var=var(+-*/^)var
    var=var
    var=number
    var=ABS(var)
    var=RND
  ACCEPT AT(row,column):var
  BREAK
  CALL CHAR(character code,Z$())
  CALL CLEAR
  CALL COLOR(row,column,color code)  (MULTICOLOR)
  CALL COLOR(code,Z$())              (BIT-MAP)
  CALL JOYST(key unit,x,y)           S=status
  CALL KEY(key unit,return var,status var)
  CALL MAGNIFY(magnification factor)
  CALL SCREEN(color code)
  DISPLAY AT(row,column):Z$()
  END
  FOR control var=initial var TO limit STEP inc.
  GOSUB line number
  GOTO line number
  IF relational expression THEN line number
  LET B=(CPTCP,CPTVD,VDTCP)*address*address*length
  LET B=(CPTS,STCP)*address*ZZ(var)
  LET B=(GRAPM,BITMM,TEXTM,MULTM,RESET
  PRINT ,,,.....
  RANDOMIZE /seed/
  RETURN
```

APPENDIX B

EFFECTS OF W() ON SCREEN DISPLAY (GRAPHICS MODE)

0	
	Characters on the Screen (768 positions)
767	
768	
	Sprite Dot-Row, Dot-Column, Pattern and Color (28 Sprites)
879	
880	
	Don't use these
1023	
1024	
	Changing these will change the Patterns (112 Characters)
1919	
1920	
	Sprite Velocities
2047	
2048	
	Character Colors (14 sets of 8 Characters)
2079	
2080	
	No effect on the screen
13567	
13568	
	Lost if disk used
16383	

APPENDIX C

ERROR MESSAGES

1. COMPILING ERRORS

BAD LINE NUMBER
- Lines are not numbered in increments of ten.
- Line number less than 2400 or greater than 9190.

BAD STATEMENT
- Invalid statement found.

MEMORY FULL
- There is no more room for object codes.

BAD VARIABLE
- An illegal variable name.
- A number is less than 0 or greater than 31.

BAD VALUE
- Specified number is not an integer from -32768 through 32767.

2. RUNNER ERRORS

Number	Description
1	BAD ROW VALUE
2	BAD COLUMN VALUE
3	OVERFLOW. Overflow in an arithmetic operation.
4	RETURN WITHOUT GOSUB
5	NEXT WITHOUT FOR
6	TOO MANY FOR-TOS. More than five loops were left open and an attempt to open a sixth one was made.
7	TOO MANY GOSUBS. The number ofGOSUBs found exceeds the number of RETURNs found by 11.
9	BAD CODE. Unrecognizable code found by the RUNNER.
11	BAD VALUE. Attempt to make an item have an unacceptable value. For example, trying to make an integer single variable less than -32768 or greater than 32767, a key unit less than zero or greater than 5, etc.
12	BACKPOINT

If an error occurs, the variable ERR is the error number, otherwise it is zero. The variable LINE is the number of the line where the error occurred plus 10. Also, the variables C through V are passed to EXTENDED BASIC as the array VAR(). Thus , VAR(1) will have the value of C, VAR(2) the value of D and so on up to VAR(20), the value of V.

APPENDIX D

TEXT MODE

In TEXT mode, the display is 40 columns by 24 rows. You can not use sprites. If you want switch back to the GRAPHICS mode you have to save the values of the array W(), from 768 through 959, and restore them before or after swithcing back to GRAPHICS mode. You have to do this for the other modes too, if your program changes these values. You can switch to TEXT mode with the statement LET B=TEXTM.

The colors of all the characters are the same. The background color is transparent. The foreground and screen colors are set with the statement

CALL SCREEN(code)

The code is the foreground color code times sixteen plus the screen color code minus 17. For example, the statement

CALL SCREEN(27)

will make the screen black and the characters light yellow (16*2+12-17=27). You can define the characters as in GRAPHICS mode, but the last dot of each row will be ignored.

The program discussed below is an example of the TEXT mode application. Actually, it is a rudimentary text editor. It edits a text of 100 lines of 80 characters each. It assumes that W(3000) through W(10999) are the character codes plus 96. Part of the text is displayed on the screen which can be scrolled up, down and sideways. The program is included after the COMPILER, so you can try it right away. After the COMPILER and the RUNNER are loaded, compile

from line 2400 through 3280. Then link to line 2400. Use the arrow keys to move and any other key to type. When the cursor reaches the screen borders, scrolling is automatically done. Pressing ERASE (FCTN 3) returns control to EXTENDED BASIC.

It works as follows. Line 2400 clears the subroutine call and FOR-TO nesting stacks of the compiled program. Lines 2410 through 2470 save W(768) through W(959). Line 2480 sets the display to TEXT mode. K points to the screen position and L points to the corresponding position on the text. Line 2640 stores the character under the cursor. Lines 2650 through 2720 accept one character and choose corresponding action. Lines 2740 through 2760 move one line up on the text. Lines 2800 through 2820 make the cursor advance one space to the right. Lines 2890 through 2910 move the cursor to the left. Lines 2980 through 3000 move the cursor down. Lines 3090 through 3180 scroll the screen either way. Line 3200 switches back to GRAPHICS mode. Lines 3210 throguh 3270 restore W(768) through W(959). Line 3280 returns control to EXTENDED BASIC.

Notice that line 3140 moves 40 consecutive characters to CPU RAM, and that line 3180 moves the 24 lines to the screen.

The scrolling can be made faster by making a few changes, like storing the text in the array X(). Of course, to make the editor more versatile, more operations, such as deleting and inserting, have to be included.

APPENDIX E

BIT-MAP MODE

In this mode you can define up to 768 characters. You can also use sprites, but their velocities will be always zero - in other words, they don't move automatically. The screen is 24 rows by 32 columns as in GRAPHICS mode. Each character can have up to 16 different colors. To define a character just use the statement

```
CALL CHAR(code,Z$())
```

where code can be from zero through 767. W(6144) through W(6911) are the characters displayed on the screen. The screen position is 6144+(row-1)*32+col-1. The screen is divided into three sections of 8 rows each. Characters 0 through 255 can be diplayed only in the first 8 rows, by simply making the corrsponding W() equal to the character code. For example, to display character 4 in the second row and third column, make W(6178) equal to 4 (since 6144+(2-1)*32+3-1=6178).

Characters 256 through 511 can only be displayed on rows 9 through 16, by making W() equal to the character code minus 256. For instance, to display character 300 on row 10 and column 5, make W(6436) equal to 44.

Characters 512 through 767 can only be displayed in rows 17 through 24, by making W() equal to the code minus 512. For example, to display character 600 on row 17 and column 1, make W(6656) equal to 88. See the table below for the effects of W() on the screen display.

To assign the sixteen colors to a character use the statement

```
CALL COLOR(code,Z$())
```

where code can be from 0 through 767. The string Z$() specifies the colors in a manner similar to the way it specifies shapes. The first character of the string Z$() specifies the color of the dots that are on in the top eight dots. The second character of Z$() specifies the color of the dots that are off in the same eight dots. The color code is now from 0 through F instead of from one throug 16. For example, black is one instead of the regular 2, and white is F instead 16. Therefore, for instance, to set the top eight dots of a given character, dots on light green and dots off dark yellow, the first two characters of Z$() must be 3 and A (corresponding to 4 and 11). The following two characters in Z$() define the color of the next 8 dots below and so on. For example, if Z$(F)="1010101010101010" is used to assign a color to a character, all the dots on will be black and all the dots off will be transparent.

The program example below fills the screen with a character, defined by Z$(1) for shape and Z$(2) for color. To try it, type it in and insert in line 1210 of the COMPILER the expressions for Z$(1) and Z$(2). For example

```
1210 Z$(1)="F0F0F0F0F0F0F0F0"::Z$(2)="0123456789
ABCDEF"::CALL LINK("RUNNER",Y(),Z$(),LINE,ERR,VAR())
```

Then, type the program, compile and link to it.

```
2400 LET B=RESET
2410 LET B=BITMM
2420 C=1
2430 F=2
2440 D=0
2450 E=256
2460 FOR I=1 TO 3
2470 CALL CHAR(D,Z$(C))
2480 CALL COLOR(D,Z$(F))
2490 D=D+E
2500 NEXT I
2510 C=6144
2520 D=6911
2530 FOR C=C TO D
2540 W(C)=0
2550 NEXT C
2560 CALL KEY(5,K,S)
2570 IF S=0 THEN 2560
2580 LET B=GRAPM
2590 END
```


It works as follows. Line 2410 sitwches to BIT-MAP mode. Lines 2420 through 2500 define the shape and colors of characters 0, 256 and 512. Lines 2510 through 2550 make W(6144) through W(6911) (all the creen positions) zero. This means that the first eight rows contain character zero, the next 8 rows character 256 and the next 8 character 562. Lines 2560 and 2570 stop the program until any key is pressed. Line 2580 switches back to GRAPHICS mode. Line 2590 returns control to EXTENDED BASIC.

EFFECTS OF W() ON SCREEN DISPLAY
(BIT-MAP MODE)

0	
	768 Character definitions (each takes 8 values)
6143	
6144	
	Characters on the first eight rows of the screen
6399	
6400	
	Characters on rows 9 through 16
6655	
6666	
	Characters on rows 17 through 24
6910	
6911	
	No effect on the screen
8191	
8192	
	768 Character colors (each color definition takes eight values)
14335	
14336	
	No effect on the screen Lost if disk is used
16383	

APPENDIX F

MULTICOLOR MODE

In this mode you can use sprites. The display is divided into 48 rows of 64 boxes. Each of the 3072 boxes has one color. To set the color of a box use the statement

```
CALL COLOR(row,column,color code)
```

For example, the statement

```
CALL COLOR(10,20,15)
```

will set to white the box at row 10 and column 20.

The table below shows how the W() values affect the screen display in this mode.

EFFECTS OF W() ON SCREEN DISPLAY
(MULTICOLOR MODE)

0	
	Don't use these
767	
768	
	Sprite dot-row, dot-column, patteern and color
879	
880	
	Don't use these
1919	
1920	
	Sprite velocities
2047	
2048	
	Don't use these W(2048) through W(2079) must be saved if you want to keep the GRAPHICS character colors
3583	
3584	
	No effect on the screnn W(13568) through W(16383) are lost if disk is used
16383	

APPENDIX G

MEMORY USE

In order to make better use of the ASBCS, you will have to become familiar with a few aspects of the memory organization of your computer, specifically with the segments of memory where your compiled program and data are stored.

Think of the computer memory as a series of labelled boxes containing informaion. The box's label is its address and the information inside is its content value. Both the address and the content value are numbers. Each of these boxes is called a byte. For example, if we say that the byte at address 10000 has a value of 7, this means thatthe box labelled 10000 has the number 7 inside.

Therefore, for our purposes, the computer memory is a series of bytes. You will only have to be concerned with the Random Access Memory (RAM). This is the part of memory which can be written to, or read from, by any program. In other words, you can change at will the content value of each byte. There are two types of RAM in your computer, one is the Central Processing Unit RAM (CPU RAM) and the other is the Video Display Processor RAM (VDP RAM). Two segments of CPU RAM are in the Memory Expansion, one from adddress 8192 to 16383 (8192 bytes) and the other from address -24576 to -1 (24576 bytes). The VDP RAM is in the console from address 0 to 16383 (16384 bytes).

The CPU RAM is used in the following manner. The RUNNER program is loaded on CPU RAM segment from address 10272 to 12973. The code numbers of the compiled program are stored from CPU RAM addresses 12974 up to 16373 (up to 3400 bytes). (Since each BASIC statement is compiled as five bytes, you can compile up to 680 BASIC lines.)

Memory Expansion CPU RAM segment from -24576 to -1 contains the main part of your EXTENDED BASIC program and free space which you can use to store data.

The VDP RAM is used in the following manner. The lower addresses contain the information that controls graphics. The higher addresses contain the string variables of your Extended Basic program.

Use the SIZE command to find out what memory spaces you can use. The VDP RAM space from address zero, through the address equal to the amount of "BYTES OF STACK FREE", is available for your use. The CPU RAM segment that you can use starts at address -24576. The number of bytes of the segment is the number of bytes given by the SIZE command as "BYTES OF PROGRAM SPACE FREE". Making changes on the memory content outside these segments will produce unpredictable results.

The elements of the array W() are the values of the bytes in VDP RAM. The value of a byte is an integer number from zero through 255. Two consecutive bytes, the first byte having an even address, are called a word. The value of a word can be an integer from -32768 hrough 32767. The variables from C through V are words stored at addresses from 10668 through 10707. C is the value of the word at address 10668, D is the word at 10670 an so on. V is the word at 10706.

The elements of X() are also words and are stored starting at address -24576. Thus, W(0) is at addresses -24576 and -24575, W(1) at -24574 and 24573 and so on until the end of the available space.

APPENDIX H

WORD AND BYTE OPERATIONS

Let's call the first and second byte of a word, B1 and B2. Let's call the value of a word WV. B1,B2 and WV are related in the following fashion

WV=B1*256+B2 if B1*256+B2<32768 and
WV=B1*256+B2-65536 otherwise.

For example, if, in a word, B1 and B2 are 100 and 200, WV is 25800. If B1 and B2 are both 200, WV is -14136.

This relation can be used to move information between VDP RAM (W()) and CPU RAM (X()). For example, if you want B1 and B2 of X(0) be equal to the W(0) and W(1), you can do this

```
C=0
D=-24576
LET B=CPTVD*0*D*2
```

As an application of these relations, consider the following example. Suppose that you want to place the character 65 in all positions of the screen. All you have to do is make W(0) through W(767) (all the screen positions) equal to 161 (65+96). You can do this by the following lines

```
2400 LET B=RESET
2410 D=767
2420 FOR C=0 TO D
2430 W(C)=161
2440 NEXT C
2450 END
```

The same operation will be done faster in EXTENDED BASIC!. The reason being that assigning

values to W(), which is equivalent to moving bytes to VDP RAM, is relatively slow. The same task can be accomplished faster by moving all the information from CPU RAM to VDP RAM as a block. The example below does just that:

```
2400 LET B=RESET
2410 D=383
2420 FOR C=0 TO D
2430 X(C)=-24159
2440 NEXT C
2450 C=-24576
2460 D=768
2470 LET B=CPTVD*0*C*D
2480 END
```

Lines 2410 through 2440 set the bytes from CPU RAM -24576 through -23809 (768 bytes) equal to 161 (since 161*256+161-65536=-24159). Line 2470 moves the 768 bytes to the screen positions of VDP RAM.

The capability of moving blocks of information around is what gives a program speed. The more statements of this type you use, the faster the resulting action will be.

Please make the following changes in the COMPILER program:

LINE	CHANGE	TO
130	"COSAOLLIST"	"COSALOLIST"
140	"SINTAX"	"SYNTAX"
1810	32767	2170
2240	32767	390

We apologize for the inconveniences and thank you for your understanding.